

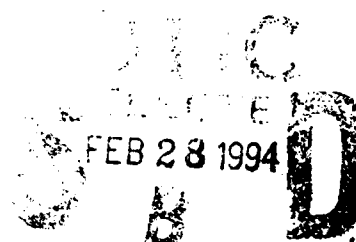


NRL/FR/5540--93-9586

# Consistency Checks for SCR-Style Requirements Specifications

CONSTANCE L. HEITMEYER  
BRUCE G. LABAW

*Center for Computer High Assurance Systems  
Information Technology Division*



December 31, 1993



# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE December 31, 1993		3. REPORT TYPE AND DATES COVERED Final	
4. TITLE AND SUBTITLE Consistency Checks for SCR-Style Requirements Specifications				5. FUNDING NUMBERS PE - 601153N PE - 62234N PE - 603794N	
6. AUTHOR(S) Constance L. Heitmeyer and Bruce G. Labrow					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5320				8. PERFORMING ORGANIZATION REPORT NUMBER NRL/FR/5540-93-9586	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Command, Control & Ocean Surveillance Center RDT&E Division San Diego, CA 92152  Chief of Naval Research, Code 882, Arlington, VA				10. SPONSORING/MONITORING AGENCY REPORT NUMBER  COMSPAWARSYSCOM Code 331-1A, Arlington, VA	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words)  This report describes a class of software tools that check formal requirements specifications for consistency with a requirements model. The model, which describes properties of requirements specifications based on the SCR (Software Cost Reduction) approach to requirements, is summarized. Two experiments are described in which condition tables and mode transition tables in an updated version of the A-7 requirements document were checked for selected properties using tools we developed. The significant number of errors found by the tools is summarized. Tool-based techniques and manual techniques for performing consistency checks are compared, and several additional consistency checks, derived from the formal model, are identified. Our work is compared with related work by Parnas and SRI concerning the automated checking of tabular specifications. Conclusions are presented concerning the utility, cost, and scalability of tool-based consistency checking.					
14. SUBJECT TERMS Software requirements Consistency check Formal model				15. NUMBER OF PAGES 16 16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

## CONTENTS

INTRODUCTION . . . . .	1
BACKGROUND . . . . .	2
Constructs for Requirements Specification . . . . .	2
Summary of the Formal Model . . . . .	3
Two Definitions from the Model . . . . .	3
RESULTS OF APPLYING CONSISTENCY CHECKS . . . . .	5
Checks on Condition Tables . . . . .	6
Checks on Mode Transition Tables . . . . .	6
Tool-based vs Manual Checks . . . . .	7
Related Work . . . . .	7
CHECKS DERIVED FROM THE MODEL . . . . .	9
Syntactic . . . . .	9
Table-Specific . . . . .	9
Non-Table-Specific . . . . .	9
CONCLUSIONS AND FUTURE WORK . . . . .	10
ACKNOWLEDGMENTS . . . . .	10
REFERENCES . . . . .	10

Accession For

NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	

R -

Dist. 100-1000

Availability Codes

Dist. 100-1000

A-1

# CONSISTENCY CHECKS FOR SCR-STYLE REQUIREMENTS SPECIFICATIONS

## INTRODUCTION

The significant human effort required to detect and correct software errors accounts for a large portion of software costs. Many software errors are introduced during the early stages of software development—during requirements definition and functional analysis. However, most are not discovered until much later—during coding, testing, and initial operational use of the software [1]. Unfortunately, the later they are discovered, the more expensive software errors are to correct. Errors discovered late can cost up to one hundred times as much to fix as errors discovered during the requirements stage [2,3].

To reduce software costs and to increase software quality, new methods are needed that help developers detect and correct errors early in the software development process, especially during the requirements stage. Our interest is in *formal methods*, i.e., mathematically based techniques useful in developing computer systems. Formal methods can provide a sound basis for building software tools that analyze software requirements specifications for errors.

An important question is what form these tools should take. To help answer this question, we are developing a prototype toolset for constructing and analyzing software (and system) requirements specifications. Our toolset includes tools that help developers generate *formal* requirements specifications, tools that use the formal specifications to simulate system operation, and “verification” tools, i.e., tools to help verify that the specifications have selected properties.

Three different classes of verification can be applied to requirements specifications. One class compares the requirements specifications with a refinement. The refinement can take many forms, such as a set of software design specifications, as in Moore’s verification of the abstract voice transmitter [4], or a set of code specifications, as in the recent Canadian effort to certify the Darlington nuclear plant shutdown systems [5]. A second class of verification checks requirements specifications for selected application properties. For example, in a railroad crossing system, a certain property must hold in every state: the crossing gate must be down if a train is in the crossing [6]. The preceding is an example of a logical property. Other important types of application properties are security properties (e.g., [7]) and timing properties (e.g., [8]).

A third class of verification, which we call *consistency checking*, is the subject of this report. A consistency checker tests the consistency of requirements specifications with a formal model. The model defines the properties of a large class of requirements specifications. Although the properties tested by a consistency checker are usually quite simple (e.g., check that a total function  $F$  is defined everywhere in  $F$ ’s domain), the number of times a property must be checked in a set of requirements specifications can be very large, and thus human reviewers may spend considerable time and effort verifying that the specifications have the properties. In fact, in the certification of the Darlington nuclear plant shutdown systems, Parnas has observed that the “reviewers spent too much of their time and energy checking for simple, application-independent properties” (such as checking for domain coverage) which distracted them from the “more difficult, safety-relevant issues” [9]. Tools that automatically perform such checks can save reviewers considerable time and effort. Moreover, they can detect errors often overlooked by human reviewers (see below).

In this report, we

- introduce our formal requirements model, which is based on the SCR (Software Cost Reduction) approach to requirements;
- identify a set of consistency checks derived from the model; and
- describe two experiments we conducted to determine the utility of automated consistency checking.

Our experiments, which checked the consistency of the software requirements specifications for a Navy avionics system, detected a significant number of errors despite the fact that the specifications had previously undergone comprehensive checks by human review teams. These results provide convincing evidence of the utility of automated consistency checking.

## BACKGROUND: FORMAL REQUIREMENTS MODEL

The SCR approach to requirements specification is the basis for our requirements model. Developed originally by Heninger, Parnas, and Shore [10-12] to specify software requirements, the approach was recently extended by Parnas and van Schouwen to describe system-level requirements [13,14]. Major goals of the SCR approach are to increase the precision and reduce the implementation bias of requirements specifications [15]. To achieve scalability, the SCR approach uses a tabular notation that produces concise yet readable requirements specifications. Recently, the SCR requirements approach was used in the certification of the Darlington systems cited above.

Underlying the SCR approach to requirements specification is a finite-state machine model that describes the required relation between the computer system of interest and its environment. Our goal in [16] is to make that model explicit so that we have a sound basis for building tools. The model is built out of a few basic SCR constructs, including monitored and controlled state variables, system modes, terms, conditions, and events. In this section we describe these constructs informally, introduce our formal requirements model, and provide formal definitions of two SCR tables to illustrate the model. In the discussion below, we refer to monitored and controlled state variables, system modes, and terms as *state variables*.

### Constructs for Requirements Specification

The objective of a requirements specification is to capture all acceptable implementations of a computer system [17]. To the extent feasible, the specifications should be abstract, describing only the externally visible behavior required of the system. Two constructs that help make the specifications abstract—monitored and controlled state variables—represent the entities of interest in the system's environment [13,14]. A *monitored* state variable represents an environmental entity that can cause the system to take some action. Changes in the monitored variables provide stimuli to the system. The *controlled* state variables represent environmental entities that the system changes in response to the stimuli. For example, in a control system that monitors water temperature and sounds an alarm when the temperature exceeds some threshold, water temperature would be represented as a monitored state variable, the alarm as a controlled state variable.

Other useful constructs for specifying requirements are conditions and events. A *condition* is a predicate about the system that is true (or false) for some measurable, nonzero time interval. Conditions are predicates on state variables; in the control system described above, " $\text{WaterTemp} \leq \text{Threshold}$ " is an example of a condition. A *primitive event* occurs when a state variable changes value. A special primitive event called a *monitored event* occurs when a monitored state variable changes value. In the control system above, a change in water temperature that exceeds the specified threshold marks the occurrence of a monitored event (given that the temperature was not greater than the threshold before the change). The occurrence of this event would cause the system to sound an alarm.

System modes and terms help simplify the specifications. A *mode class* is a state machine, whose states are called *system modes* (or simply *modes*) and whose transitions are triggered by events. Complex systems are defined by more than one mode class, operating in parallel. At any given time, a system must be in exactly one mode from each mode class. In the control system example, two system modes can be identified, "normal" mode and "hazardous" mode. A high water temperature, a hardware fault, or some other event can cause the system to move from "normal" mode to "hazardous" mode. Clearly, the required system behavior in normal mode will be different from system behavior in the hazardous mode. If at any given time, the control system must be in either normal mode or hazardous mode, then these two modes form a mode class. Each *term* is a function of monitored state variables, modes, or other terms. In the control system example, the system may use three sensors to measure water temperature. To express the control system's requirements, a term named **MajorityHigh** may be defined that has the value *true* if two or more sensors indicate a temperature above the threshold, *false* otherwise.

### Summary of the Formal Model

Reference 16 contains an initial version of our formal requirements model. Like the model introduced by Faulk [18], our model describes a computer system as a finite-state automaton  $(S, s_0, E, T)$ , where  $S$  is a set of system states,  $s_0$  is an initial state,  $E$  is an input alphabet consisting of monitored events, and  $T$  is a system transform describing the changes that monitored events trigger in the system state. In the model, the system state  $s$  is a set of ordered pairs,  $s = \{(r, v)\}$ , where  $r$  is the name of a monitored or controlled state variable, a mode class, or a term, and  $v$  is a value. The state  $s$  is treated as a function, where  $s(r) = v$  iff  $(r, v) \in s$ .

A fundamental assumption of our initial model is that the system transform  $T$  is a function. For some applications, this assumption, which forces the requirements specifications to be deterministic, is overly restrictive and can lead to overspecification of the requirements. To avoid this problem, new versions of the model will allow nondeterminism. However, checking specifications for nondeterminism still has utility. Instances of nondeterminism in the specifications should be brought to the user's attention, so that he/she can confirm that the nondeterminism is intentional, not an error.

To define required system behavior, SCR specifications (such as the A-7 specifications [11] and the Water Level Monitoring System [14]) use a collection of tables, among them selector tables, condition tables, event tables, and mode transition tables. Our requirements model uses functions to define the meaning of these tables. The functions defining selector tables and condition tables describe constraints on the system state. A selector table maps a mode to an *output* (e.g., a controlled state variable) in the same state; a condition table maps a mode and a condition to an output in the same state. The functions that define event tables and mode transition tables describe constraints on the system's state transitions. An event table (mode transition table) maps a mode and an event to an output (a mode) in the new state.

### Two Definitions from the Model

We present two definitions from our initial model. The first definition describes the meaning of condition tables; the second the meaning of mode transition tables. In the definitions,  $TY$  is a function that maps a state variable or mode to its data type,  $r'$  is the name of a mode class, and  $M_r$  is the set of values of modes in that mode class (i.e.,  $TY(r') = M_r$ ).

#### Condition Tables

In an SCR requirements specification, each condition table defines a controlled state variable or a term as a function of modes and conditions. All condition tables must satisfy certain properties, including:

- **Coverage Property.** The disjunction of the conditions in a row is true.<sup>1</sup> This property forces the function to be total.
- **Mutual Exclusion Property.** The conjunction of any pair of conditions in a row is false. Otherwise, the definition does not describe a function.

Below, we show one possible format for a condition table in an SCR specification and present a function which describes the table's meaning. The function definition consists of the table's syntax,<sup>2</sup> its semantics, and a set of constraints, including the two properties listed above. The purpose of the constraints is to ensure that the definition conforms to the formal model.

Modes	Conditions			
$m_1$	$c_{1,1}$	$c_{1,2}$	$\dots$	$c_{1,p}$
$m_2$	$c_{2,1}$	$c_{2,2}$	$\dots$	$c_{2,p}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$m_n$	$c_{n,1}$	$c_{n,2}$	$\dots$	$c_{n,p}$
$r$	$v_1$	$v_2$	$\dots$	$v_p$

**Syntax:** A condition table is a function  $F_r(m_j, c_{j,k}) = v_k$ , where  $r$  is a controlled state variable or term,  $m_j$  is a mode in mode class  $M_1$ ,  $c_{j,k}$  is a condition, and  $v_k$  is a value.

**Semantics:** In state  $s$ ,  $s(r') = m_j$ , there exists  $k$ :  $c_{j,k}$  is *true* in state  $s$ , and  $F_r(m_j, c_{j,k}) = v_k \rightarrow s(r) = v_k$ .

**Constraints:**

- (1) For all  $j$ ,  $\bigvee_{k=1}^p c_{j,k} = \text{true}$  (Coverage Property).
- (2) For all  $j, k, k', k \neq k'$ :  $c_{j,k} \wedge c_{j,k'} = \text{false}$  (Mutual Exclusion Property).
- (3) The modes  $m_j$  are distinct and  $\bigcup_{k=1}^n m_j = M_1$ .
- (4) For all  $j$ ,  $v_j \in \text{TY}(r)$  (Type Checking).

### Mode Transition Tables

An SCR requirements specification contains one mode transition table for each mode class. Each mode transition table defines a new mode as a function of the current mode and a conditioned event. In our formal model, a *conditioned event* is an ordered pair  $(e, c)$ , where  $e$  is a primitive event and  $c$  is a condition. A conditioned event  $(e, c)$  occurs in state  $s$  when event  $e$  occurs in state  $s$  and condition  $c$  is true in state  $s$ .<sup>3</sup> As with condition tables, functions defined by mode transition tables must satisfy specified constraints. For example, each event and current system mode must be mapped to a unique new system mode (i.e., the next-state mapping is deterministic).<sup>4</sup> Below, we show a possible format of a mode transition table in an SCR specification and present a function defining the table's meaning.

<sup>1</sup>As P. Clements has observed, the Coverage Property may not hold when dependencies among the state variables are referred to in the conditions [19].

<sup>2</sup>This definition requires all modes appearing in a condition table to belong to the same mode class. The A-7 requirements document contains examples of condition tables in which the modes do not all belong to the same mode class.

<sup>3</sup>In the SCR approach, the simple event represented as  $\text{GT}(A)$  means "condition A becomes true"; the simple conditioned event represented by  $\text{GT}(A) \text{ WHEN } B$  means "condition A becomes true when condition B is true." The simple event  $\text{GF}(A)$  can be represented as  $\text{GT}(\neg A)$ .

<sup>4</sup>This is required for the function to be well-defined. Note that this property is identical to the Mutual Exclusion Property. For historical reasons, we refer to this property as Determinism when discussing mode transition tables.

Current Mode	Set of Conditioned Events	New Mode
$m_1$	$E_{1,1}$ $E_{1,2}$ ... $E_{1,J_1}$	$m_{1,1}$ $m_{1,2}$ ... $m_{1,J_1}$
$m_2$	$E_{2,1}$ $E_{2,2}$ ... $E_{2,J_2}$	$m_{2,1}$ $m_{2,2}$ ... $m_{2,J_2}$
...	...	...
$m_p$	$E_{p,1}$ $E_{p,2}$ ... $E_{p,J_p}$	$m_{p,1}$ $m_{p,2}$ ... $m_{p,J_p}$

**Syntax:** The mode transition table for the mode class associated with  $r'$  is a function  $F_{r'}(m_k, E_{k,j_k}) = m_{k,j_k}$ , where  $m_k$  and  $m_{k,j_k}$  are modes in mode class  $M_i$ , and  $E_{k,j_k}$  is a set of conditioned events.

**Semantics:** Given states  $s$  and  $s^*$ ,  $s(r') = m_k$ , there exists  $j_k$ : conditioned event  $e \in E_{k,j_k}$  occurs in state  $s$ , and  $F_{r'}(m_k, E_{k,j_k}) = m_{k,j_k} \rightarrow s^*(r') = m_{k,j_k}$ .

**Constraints:**

- (1) The modes  $m_k$  are distinct.
- (2) For all  $k, j_k, j'_k, j_k \neq j'_k$ :  $(e, c) \in E_{k,j_k}, (e', c') \in E_{k,j'_k}$ ,  
 $m_{k,j_k} \neq m_{k,j'_k} \rightarrow e \cap e' = \emptyset$  OR  $c \wedge c' = \text{false}$  (Determinism).<sup>5</sup>
- (3)  $m \in M_i \rightarrow \exists k, 1 \leq k \leq p: m = m_k \vee \exists k, 1 \leq k \leq p, \exists j_k, 1 \leq j_k \leq J_k: m = m_{k,j_k}$   
(each mode in the mode class  $M_i$  is in either  $F_{r'}$ 's domain or its kernel).

## RESULTS OF APPLYING CONSISTENCY CHECKS

We conducted two experiments to evaluate the utility of applying consistency checks to requirements specifications. In the experiments, software tools were constructed and used to check tables in an updated version [12] of the A-7 requirements document [11]. By means of a collection of tables, this document specifies the required behavior of the A-7E aircraft's Operational Flight Program (OFP), a program containing approximately 16,000 lines of very tight assembly language code. The new document [12] corrects several errors in the original and uses a tabular format, designed by Faulk [18], to specify mode transitions. Advantages of the new format are improved readability and increased amenability to automated analysis.

The checks that we applied to condition tables and mode transition tables and the significant numbers of errors that the tools uncovered are described below. We also compare tool-based consistency checking with manual consistency checking and discuss the relationship between our work and other recent work analyzing SCR-style specifications. Table 2 below refers to *input data items* (indicated by the notation /.../). Whereas monitored state variables represent *system* inputs, input data items represent *software* inputs. In the control system described above, water temperature would be denoted by a monitored state variable, the reading of a sensor that measures water temperature by an input data item.

<sup>5</sup>This check for nondeterminism does not cover all cases. For example, it cannot detect nondeterminism that occurs because of dependencies among events. An instance of this occurs when two events are both triggered by a third event but are also both triggers of change (conditioned events) in a mode table. Although our tool did not detect it, automated detection of this class of nondeterminism is possible.

### Checks on Condition Tables

In the first experiment, we constructed a tool that checked all of the condition tables in Ref. 12 for the two properties described previously—the Coverage Property and the Mutual Exclusion Property. All of the condition tables in the specifications—36 tables consisting of 98 rows—were tested. Our tool found 19 errors. Seventeen of these, distributed over 11 tables, proved to be legitimate errors. Table 1 shows the four types of errors that were found, the number of instances of each error, and a description of the error.

Our tool only checked the variables in condition tables that were Boolean.<sup>6</sup> Therefore, it did not have enough information to determine that two of the flagged errors were not truly errors. Both cases involved three values, namely, !Mark!, !Dest 0!, and !Dest 1-9!, which describe combinations of settings of two input devices. These three values are all the possible values and do not overlap. We confirmed by hand that the two rows containing these terms are correct.

Table 1 — Errors Detected in the A-7E Condition Tables

ERROR	INSTANCES	EXPLANATION
Slewing Variable	9	While behavior for 2 of the 3 values of the variable Slewing, namely, !Before slewing! and !After slewing!, is specified, behavior for third value, !During slewing!, is missing.
*GrTest*	4	*GrTest* is a mode with submodes. Some tables do not specify behavior for all the *GrTest* submodes.
Steering Phases	3	In early document, 3 values were used to describe phases of steering to a target. In revised document, 4 values used but some tables have not been updated.
Application-Specific Knowledge	1	"!OTS! OR !Range to RMAX!<0" and "NOT (!range! to !target! <= 10 miles)" do not form a partition.

### Checks on Mode Transition Tables

In a second experiment, we built a tool that checked all of the mode transition tables defined in Ref. 12 for nondeterminism. The A-7 specifications contain three mode classes, with 46 different modes distributed among them (18 modes in the first mode class, 7 modes in the second, and 21 modes in the third). The tool checked 688 rows, each row containing a simple event or the conjunction of an event and one or more conditions. Thirty-three transitions were found to be nondeterministic. Although many of these transitions are undoubtedly errors, a few probably are not, since some of the detected events may be impossible.<sup>7</sup>

<sup>6</sup>Of the 44 variables used in the condition tables, 33 variables were of type Boolean or were converted by hand to type Boolean. (The latter were either modes or enumerated types with two values.) The remaining 11 variables, which were either continuous or enumerated with more than two values, were used in relational expressions. The relational expressions were also converted manually into Boolean variables.

<sup>7</sup>Another class of nondeterminism is nondeterminism involving simultaneous events. Although our tool was not designed to detect this class of nondeterminism, automatic detection of nondeterminism caused by simultaneous events is feasible.

To illustrate this class of errors, Table 2 shows two instances of nondeterminism detected by the tool in the mode transition table for the Alignment, Navigation, and Test Mode Class. (Table 2 shows only a small part of the mode transition table. In Ref. 12, the complete table is distributed over 14 pages.) The transitions that are marked allow the system to transfer from Inertial Mode (\*I\*) to either Airborne Alignment Mode (\*Airaln\*) or to Doppler Inertial Mode (\*DI\*). An example of an event that could trigger either transition is

```
αT(!Doppler up!) WHEN NOT !CA stage complete! ^ NOT !present position entered!
    ^ !latitude! > 70 degrees ^ NOT!latitude! > 80 degrees ^ /IMSMODE/    $Gudal$
```

### Tool-based vs Manual Checks

Prior to its publication, the A-7 requirements document underwent several careful reviews based on the techniques described in Ref. 19. The reviewers were NRL computer scientists (including one of the authors) and engineers at the Naval Air Warfare Center in China Lake, California, who maintained the OFF. As stated above, the reviewers overlooked many significant errors that our tools detected.

That errors were detected should not diminish the credit due the reviewers; they did an excellent job given the large volume and complexity of the requirements data. In our view, tools, such as the ones we built, can complement the efforts of software developers. Human effort is crucial to acquiring the requirements information and translating it into a formal representation. Moreover, after errors have been detected in the specifications, human intervention is needed to correct the errors. However, once human specifiers have developed a reasonable draft of the requirements specifications, software tools provide a quick, effective means of checking that the specifications satisfy the properties of interest. Not only are tools more effective than people in detecting errors of the class described above, the availability of such tools can largely eliminate a labor-intensive task that humans find tedious and boring. (A strongly negative view of the review task was expressed by review participants in interviews conducted after the completion of the Darlington certification effort [20].)

In addition to their utility, another important feature of the tools is their low cost. In the Darlington certification effort, which was estimated to cost more than \$40M, teams of reviewers manually checked the software requirements specifications (and the code specifications) for application-independent properties, such as the Mutual Exclusion and Coverage properties described above. In addition, reviewers looked for discrepancies between the requirements specifications and the code specifications. (This is the first class of verification described in the introduction.) Clearly, tools that compare the specifications with a refinement are more complex than the tools that we built. However, that does not diminish the value of our simple tools. Parnas has observed that the "majority of the theorems that arose in the documentation and inspection of the Darlington Nuclear Plant Shutdown Systems" were simple properties and that the reviewers needed to analyze 40 kg of trivial tables for such properties [9]. Using tools like ours to do such checks should cost far less than using people.

### Related Work

Both our work and recent work by Atlee and Gannon [21] used software tools to analyze SCR requirements specifications. The two efforts differ in three respects. First, we tested the specifications for properties defined in our requirements model, whereas they tested application properties. Second, our tools were developed inhouse, whereas they used Clarke's model checker [22]. Finally, our tools checked two kinds of tables, condition tables and mode transition tables; the methods in Ref. 21 were used to check mode transition tables only.



Other related work is by Parnas and SRI, International. In a recent paper [9], Parnas describes 10 small theorems related to his tabular notation and challenges the developers of automated proof systems, such as EVES [23] and PVS [24], to use their systems to prove the theorems. Two of the theorems, the Domain Coverage Theorem and the Disjoint Domains Theorem, are slight variations of the Coverage Property and the Mutual Exclusion Property described above. SRI researchers accepted the challenge. In a recent paper [25], they describe how nine of Parnas' theorems were proven automatically using the "tec-strategy" (tec's are type-correctness conditions) of SRI's proof system, PVS [24]. That PVS can prove such theorems easily is not too surprising, since the proofs require very simple logic. What is noteworthy about the PVS experiment is that the theorems were proven automatically. Clearly, a toolset such as ours should be tied eventually to a mechanical proof system (e.g., PVS or a similar system) that encapsulates some subset of conventional logic, so we need not encode the logic ourselves. Our toolset could then use the encoded logic to check specifications for the properties of interest.

## CHECKS DERIVED FROM THE MODEL

The checks performed in our experiments were extracted from a set of consistency checks derived from our formal model. We list here a more complete set of consistency checks that can be applied to tabular specifications, organizing them into three groups: Syntactic Checks, Table-Specific Checks, and Non-Table-Specific Checks. Note that some of the syntactic checks must be applied before other checks can be applied. For example, type checking should precede a check of condition tables for the Coverage Property.

### Syntactic

These checks test the syntactic correctness of the tables:

- Expressions describing conditions and events are well-formed.
- Each table entry has the proper data type.
- Each controlled state variable, term, and mode class is defined by exactly one table.

### Table-Specific

For each type of table, there are table-specific checks. Examples include the two properties described above (Coverage and Mutual Exclusion) that condition tables must satisfy. Listed below are additional examples of table-specific checks, each applicable to mode transition tables.<sup>8</sup>

- Every mode in a mode class is in either the "current mode" column or the "new mode" column.
- Each mode class is associated with exactly one mode transition table.
- No mode is unreachable.

### Non-Table-Specific

This group includes checks that can be applied to more than one table. The second check listed includes the check on the mode transition tables described previously.

- The initial state  $s_0$  is unique. Initial values are required for all mode classes, terms, and monitored and controlled state variables.
- The transform  $T$  is deterministic. This requires checking both the event tables and the mode transition tables for nondeterminism.

<sup>8</sup>A violation of the third property is not an error. Hence, its violation should result in a warning rather than an error message.

- There are no identical modes. (Two modes are identical if they exhibit the same behavior.)<sup>9</sup>
- There are no circular dependencies that lead to contradictions. (An example of a circular self-dependency: 'If in mode  $m$  the event  $x = a$  occurs, then  $x = b$ .' Circular dependencies among several tables are even more difficult to check, especially by hand.)

## CONCLUSIONS AND FUTURE WORK

There are three conclusions from our work.

- Consistency checkers can be highly effective in detecting errors in requirements specifications. Not only can such tools find errors people miss, they can also liberate people from the unpleasant task of checking specifications for consistency.
- Properly designed tools are more cost-effective than human reviewers for doing certain types of consistency checks.
- The formal methods on which our tools are based scale up. They detected a significant number of errors in a medium-size real-world specification.

Our plans are to develop the formal requirements model further and to continue experimenting with prototype tools based on the model. Work is currently in progress to add timing, precision, and nondeterminism to the formal model. Three tools are planned, all using the engineering approach described in Ref. 26. The first tool is a consistency checker. A second tool will test the specifications for selected application properties, thus implementing the second class of verification described in the introduction. A third tool will derive simulations from the formal specifications; the definition of the system transform in Ref. 16 provides a basis for building a simulator of SCR-style requirements specifications.

We expect our formal model to provide a solid conceptual foundation for developing requirements specifications and our suite of tools to demonstrate how formal methods can improve the quality of the specifications. High-quality requirements specifications should significantly reduce the costs of software development.

## ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of our NRL colleagues, P. Clements, C. Gasarch, R. Jeffords, D. Kiskis, A. Rose, and W. Smith, and J. Gannon of the University of Maryland. C. Gasarch and A. Rose developed the tool for checking mode transition tables; P. Clements wrote the program translating the text format language of the A-7 Requirements Document into Modechart specifications; and D. Kiskis developed the tool that tested condition tables. R. Jeffords and D. Kiskis helped identify the consistency checks. Finally, P. Clements, R. Jeffords, J. Gannon, and W. Smith made valuable comments on an earlier draft.

## REFERENCES

1. C.V. Ramanmoorthy et al., "Software Engineering," *IEEE Computer* 17(10), 191-209 (1984).
2. B. Boehm, *Software Engineering Economics* (Prentice-Hall, Englewood Cliffs, NJ, 1981).
3. R. Fairley, *Software Engineering Concepts* (McGraw-Hill, New York, 1985).
4. A.P. Moore, "The Specification and Verified Decomposition of System Requirements Using CSP," *IEEE Trans. Soft. Eng.* SE-16(9), 932-948 (1990).

<sup>9</sup>A violation of this property is not an error; hence, it should result in a warning rather than an error message.

5. D.L. Parnas, G.J.K. Asmis, and J. Madey, "Assessment of Safety-Critical Software in Nuclear Power Plants," *Nuclear Safety* 32(2), 189-198 (1991).
6. F. Jahanian and D.A. Stuart, "A Method for Verifying Properties of Modechart Specifications," In *Proc., Real-Time Systems Symposium*, Huntsville, AL, December 1988.
7. C.E. Landwehr, C.L. Heitmeyer, and J. McLean, "A Security Model for Military Message Systems," *ACM Trans. Comp. Syst.* 2(3), 198-222 (1984).
8. C.L. Heitmeyer and B.G. Labaw, "Requirements Specification of Hard Real-Time System: Experience with a Language and a Verifier," *Foundations of Real-Time Computing: Formal Specifications and Methods*, A. van Tilborg and G. Koob, eds. (Kluwer Academic Publishers, Norwell, MA, 1991).
9. D.L. Parnas, "Some Theorems We Should Prove," in *Proc., 1993 Int. Conf. on HOL Theorem Proving and Its Applications*, Vancouver, BC, August, 1993, pp. 156-163.
10. K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans. Soft. Eng.* SE-6(1), (1980).
11. K.L. Heninger, D.L. Parnas, J.E. Shore, and J. Kallander, "Software Requirements for the A-7E Aircraft," NRL Report 3876, 1978.
12. T.A. Alspaugh, S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, and J.E. Shore, "Software Requirements for the A-7E Aircraft," NRL/FR/5546--92-9194, 1992.
13. D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering (Version 2)," Technical Report CRL 237, Telecommunications Research Inst. of Ontario (TRIO), McMaster Univ., Hamilton, Ont., 1991.
14. A.J. van Schouwen, "The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application for Monitoring Systems," Technical Report TR 90-276, Queen's Univ., Kingston, Ont., 1990.
15. S. Faulk and P. Clements, "The NRL Software Cost Reduction (SCR) Requirements Specification Methodology," *Proc., Fourth Int. Workshop on Software Specification and Design*, April 1987.
16. C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Tools for Analyzing SCR-style Requirements Specifications: A Formal Foundation," NRL Report in preparation.
17. C.L. Heitmeyer and J. McLean, "Abstract Requirements Specifications: A New Approach and Its Application," *IEEE Trans. Eng.* SE-9(5) (1983).
18. S.R. Faulk, *State Determination in Hard-Embedded Systems*, PhD thesis, University of North Carolina, Chapel Hill, NC, 1989.
19. P. Clements, Naval Research Laboratory, personal communication, April 1993.
20. D.L. Parnas and D.M. Weiss, "Active Design Reviews: Principles and Practices," in *Proc., Eighth Int. Conf. on Software Engineering*, 1985.
21. D.H. Craigen et al., "An International Survey of Industrial Applications of Formal Methods," NRL/FR/5546--93-9581, 1993.

22. J. Atlee and J. Gannon, "State-based Model Checking of Event-driven System Requirements," in *Proc. ACM SIGSOFT Conf. on Software for Critical Systems*, New Orleans, December 1991.
23. E.M. Clarke, E. Emerson, and A. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Prog. Lang. Systems* 8(2), (1986).
24. D.H. Craigen, "Reference Manual for the Language Verdi," Technical Report TR-91-5429-09a, Odyssey Research Associates, Ottawa, Ont., 1991.
25. S. Owre, N. Shankar, and J. Rushby, "User Guide for the PVS Specification and Verification System," draft technical report, Computer Science, SRI International, Menlo Park, CA, 1993.
26. J. Rushby and M. Srivas, "Using PVS to Prove Some Theorems of David Parnas," in *Proc., 1993 Int. Conf. on HOL Theorem Proving and Its Applications*, Vancouver, BC, 1993.
27. C.L. Heitmeyer, P.C. Clements, B.G. Labaw, and A.K. Mok, "Engineering CASE Tools to Support Formal Methods for Real-Time Software Development," in *Proc., Fifth Int. Workshop on Computer-Aided Software Engineering*, July 1992.